Demands on Software in Computer Algebra and Object oriented Solutions

by

Marc Conrad (Universität des Saarlandes)

1 Introduction

In the last ten years object oriented programming has gained an increasing importance in the design of software. However, in the context of mathematical software, there has not been so much influence as one should expect. On the commercial sector only Axiom explicitely uses object oriented features as e.g. (multiple) inheritance. A well known example of a non commercial software package in number theory which uses object oriented ideas is LiDIA [4]. But there the use of C++ as implementation language and a too narrow focus on efficiency force to many compromises. As it is pointed out by Papanikolaou [5] the basic structures of LiDIA are only object *based*.

On the other hand, it is interesting to see that in other existing computer algebra systems object oriented principles are used without even noticing by the programmers that they are object oriented. For instance SIMATH [6] is as a C-library a priori not object oriented. However, since 1988, the basic functions for matrix arithmetic have been implemented independent from the base ring. Such an implementation is polymorphic which is – as we will see in Section 2 – a basic technique in object oriented programming. An overview of object oriented strategies in LiDIA, SIMATH and other packages concerning elliptic curves can be found in [3]. Note also that currently the project CAGE [2] aims to establish an object oriented computer algebra system.

Observations as described above show the necessity for a closer look to the subject from a general, theoretical point of view. Therefore we will give a description of object oriented techniques for mathematical structures which is independent of both an implementation language and existing computer

$\mathbf{Q}(\alpha)$	
implementation	interface
data: $f(x), \ldots$ algorithms: compute $[\mathbf{Q}(\alpha) : \mathbf{Q}]$ by $\deg(f)$ $\ldots$	methods: return generating polynomial return $[\mathbf{Q}(\alpha) : \mathbf{Q}]$ $\ldots$

Figure 1: An object *algebraic number field*

algebra packages.

To find an abstract definition of an object oriented concept is not so easy as one would expect. Most treaties relate to a specific language as C++ or Java and ignore concepts which are difficult to implement in this language. A quite comprehensive, language independent description, can be found in [1]. Unfortunately, in this book there are almost no examples concerning mathematical structures. So, we describe in Section 2 the most important terms *objects*, *methods*, *inheritance*, *overriding* and *polymorphism* in giving mathematical examples.

Having answered the question on *what* object oriented techniques are, we discuss in Section 3 *why* these techniques are useful. We give there an overview on the demands on software in computer algebra as they are formulated in [5] and show how object oriented programming meets the requirements of these demands.

2 Description of object oriented techniques

In this section we will illustrate the main elements of object oriented programming by giving examples coming from mathematical structures, especially fields and polynomials.

Note that these examples are not intended to be complete concepts. On the contrary, they have been kept simple in order to point out the main ideas. A complete concept concerning elliptic curves can be found in [3].

2.1 Objects and Methods

The basic idea in object oriented programming is the idea of having objects which communicate with each other by invoking methods. At least one of the methods is used to construct the object. We call such a method a *constructor* of the object.

For instance an object *algebraic number field* $\mathbf{Q}(\alpha)$ may be constructed by an irreducible polynomial $f(x)$ which is the minimal polynomial of α. A method of $\mathbf{Q}(\alpha)$ which gives the degree $[\mathbf{Q}(\alpha) : \mathbf{Q}]$ will invoke a method of the object $f(x)$ which returs $\deg(f)$.

Each object has two faces: the *interface* and the *implementation*. The interface contains the methods and is accessible to the intended user whereas the implementation remains hidden.

So, in the example of $\mathbf{Q}(\alpha)$ we focus not how the object *algebraic number field* is internally represented or how its methods are actually performed, but which methods are appropriate do describe an algebraic number field. For instance a method which returns the generating polynomial $f(x)$ of the field would mostly be implemented in simply reading out the polynomial which is stored as data of the field. But an algebraic number field could also be constructed by the information that it is the splitting field of a (not necessarily irreducible) polynomial $g(x)$. Then the method which computes $f(x)$ will be more complicated.

2.2 Inheritance

"An object B inherits from an object A" means that the methods of A are also available in B. In addition B may have new methods. We say also that B is derived from A. For instance an object *abelian number field* may inherit the methods from the object *algebraic number field*. The abelian number

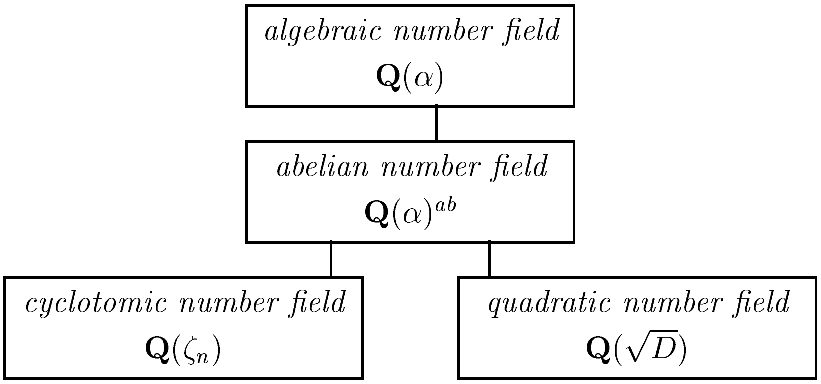

Figure 2: An object hierarchy of algebraic number fields.

field F_{ab} then might have as a new method the minimal cyclotomic field from which F_{ab} is a subfield. A collection of objects which are connected by inheritance is called an object hierarchy. Figure 2 shows such an hierarchy which realizes algebraic number fields.

The connection between objects can also be described as an "is a" relationship. B inherits from A means that B *is an* A. So, an abelian number field is a number field and a cyclotomic and quadratic field are both abelian number fields and therefore number fields.

A derived object may obtain additional constructors. There is no canonical way to construct an abelian number field by an integer. But for instance the construction of a cyclotomic field by an integer n could mean to construct the n-th cyclotomic field whereas the construction of a quadratic field by an integer n may lead to the field $\mathbf{Q}(\sqrt{n})$.

The concept of *multiple inheritance* enables an object B to inherit methods from two objects A and A'. In our example an object $\mathbf{Q}(i)$ could be a subclass of both the quadratic and the cyclotomic number field. In general, multiple inheritance often leads to subtle programming problems. E.g. note that $\mathbf{Q}(\zeta_n)$ and $\mathbf{Q}(\sqrt{D})$ can have methods with the same name and different meanings. For a detailed discussion see [1].

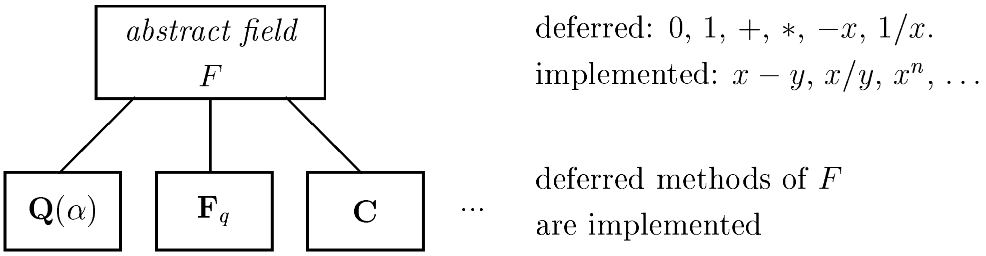

Figure 3: Deferred and implemented methods.

2.3 Overriding Methods

An important concept in object hierarchies is the possibility to *override* methods. This means that an object may reimplement a method of its parent class. We give three examples which show where overriding occurs and illustrate why it is necessary.

Concrete implementations in abstract data types: Suppose we have an object hierarchy as in Figure 3 where the abstract field F is inherited by other fields. The field F may have e.g. a method which provides the multiplication of field elements and a method which returns the 1 of the field. These methods of F cannot have an implementation (or alternatively, they have a trivial implementation returning an error value), as only their existence is known. Methods which are available but not implemented are called *deferred* methods. However, the deferred methods of the abstract field can be used to perform other arithmetical operations e.g. to implement a powering method which computes a^n for $a \in F$ and $n \in \mathbf{N}$.

A concrete field as the finite field $\mathbf{F}_q$ or the algebraic number field $\mathbf{Q}(\alpha)$ needs to override the deferred methods. By inheritance the other methods are then immediately available.

Side effects of new methods: Suppose that the method in the object $\mathbf{Q}(\alpha)$ (in the hierarchy of Figure 2) which returns $f(x)$ is implemented by simply reading out the defining polynomial from data. In the derived objects $\mathbf{Q}(\zeta_n)$ and $\mathbf{Q}(\sqrt{n})$ this method has to be reimplemented since these fields can be also defined by the integer n.

Performance improving: The multiplication routine in $\mathbf{F}_q$ may be implemented as multiplication of two polynomials over $\mathbf{F}_p$ and the reduction modulo a polynomial which generates $\mathbf{F}_q$. In the case that $q = p$ is a prime such an approach would cause much overhead. For the sake of efficiency it is therefore recommended to derive an object $\mathbf{F}_p$ of $\mathbf{F}_q$ which then overrides the multiplication of $\mathbf{F}_q$ with a faster algorithm.

2.4 Polymorphism

According to [1] a polymorphic object is any entity such as a variable or a function argument that is permitted to hold values of different types during the course of execution. This allows the construction of structures which are related to a collection of objects with similar properties. Of special interest is the situation where a polymorphic reference represents an arbitrary object of a given hierarchy.

For instance we may construct an object *polynomial ring* over an arbitrary field using a polymorphic reference to the field hierarchy. Because this reference gives access to the arithmetic of the field, the basic functionalities of a polynomial ring as multiplication, addition etc. can be implemented directly.

A more sophisticated approach is needed to implement methods for which the algorithm is different depending on the field. An example for such a situation is the factorization of a polynomial into irreducible factors. The existence of such a factorization is independent of a field whereas the concrete factorization algorithms differ considerably (cf. $\mathbf{C}[X]$, $\mathbf{F}_p[X]$ and $\mathbf{Q}[X]$). There are two main strategies to implement this:

(1) From the object *polynomial ring* we derive extra objects *polynomial ring over complex numbers*, *polynomial ring over a finite prime field*,

$F[X_1, \ldots, X_n]$:

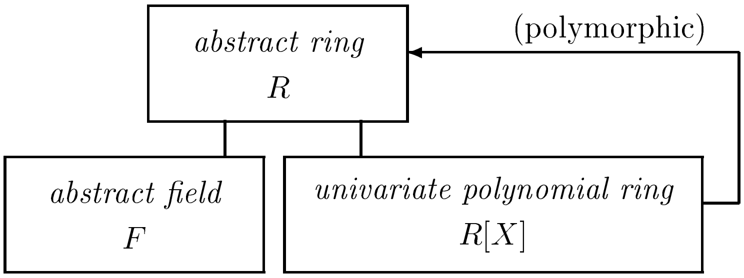

Figure 4: Multivariate polynomials by univariate polynomials.

etc. The unspecified object *polynomial ring* then defines the factorization as a deferred method which is overridden in the concrete objects.

(2) The method is fully implemented in the object *polynomial ring* switching between different fields. So, the type of the field has to be determined at run time. With this type information the appropriate algorithm can be called.

As a general rule one can say that type checking at compile time is more efficient then at run time. However the approach described in (1) has disadvantage of the necessity to define a lot of different objects with similar properties. Such a setting can make a concept unwieldy and difficult to handle. A good object oriented design will always be a compromise between the use of run type decisions and defining new objects.

We will modify now the above example of a polymorphically defined polynomial ring in order to show how powerfull polymorphism can be. Instead of the reference to a field we now assume that the reference points to an object *ring*. Because *ring* is a parent class of both *field* and *polynomial ring*, we can construct recursively a polynomial ring $R[X_2]$ over a polynomial ring $R = F[X_1]$. So, it suffices to implement an object *univariate polynomial ring* as it is show in Figure 4 to obtain a multivariate polynomial ring $F[X_1, X_2] = F[X_1][X_2]$.

3 Advantages of object oriented techniques

In Section 2 we have described object oriented techniques. Now, we will show why these techniques should be used. In [5] the demands on software in computer algebra are formulated. We give here a description of these demands and illustrate that object oriented structures meet the requirements of these demands.

1. *Efficiency:* Because managing of objects hierarchies is time consuming, efficiency is the most common argument against object oriented programming. There are two main arguments against this. First, the need for efficiency is not only given for running a program but also for *developing* and *using* a program. The advantage of well structured classes and the possibility of code reuse usually more then make up for any small losses in efficiency.

   In addition, time critical methods can be overridden in a derived object. For instance a method for adding points on an elliptic curve over a (polymorphic) field may be overridden for elliptic curves over a finite prime field since this case is important for cryptographic implementations. For other fields as e.g. number fields the time for performing the field arithmetic can overweight the time which is needed to handle the polymorphism.

2. *Reuseability:* The possibility to reuse code is one of the most powerful advantages of object oriented programming. The concept of inheritance enables the programmer to implement mathematical structures only once on the most abstract level. In the example of the field hierarchy this has the consequence that e.g. the powering method can be reused in each concrete field although there is only one implementation.

3. *Adaptability:* A software package in mathematics has to be up to date concerning the current state of research. Therefore there must be

the possibility to replace parts of systems. For instance, during the more then ten years long development of SIMATH [6] the basic integer arithmetic has been modified or replaced at least three times which was a difficult task to do. So, one cannot rely that there is a "best" integer arithmetic for all platforms and for all times. The same is true for other basic structures as memory management, aggregate types or data bases.

In an object oriented environment the objects are known to other objects only by their methods. The internal implementation is hidden and can therefore not be accessed by other objects. From this it follows that an object can safely be replaced by another one if the interface is the same. Note that this is successfully used in CAGE [2] and LiDIA [4] concerning the basic arithmetic.

4. *Extensibility:* Of course it is always possible to derive new objects from existing objects or to introduce new objects. This is even easier since existing code can be used. If there is for instance a need to work with an algebraic extension of a p-adic field $\mathbf{Q}_p(\alpha)$ we have only to override the basic routines of the abstract field in order to gain full performance, e.g. polynomial rings over $\mathbf{Q}_p(\alpha)$.

   Beside this there is also the possibility to add new objects as new parent classes by moving the appropriate code from the child class. So, we may extend the field hierarchy by inserting a new object *ring* by moving the code for addition, multiplication etc. from the object *field* to *ring* and then deriving *field* from *ring*. Note that by this procedure the interface of *field* is not changed.

5. *Portability:* Because of the ongoing progress in hardware development it is of importance that a package does not depend on a special platform. In an object oriented environment all machine depended code can (and should) be encapsulated in special objects. So, when porting the software package to another machine, only these objects have to

be changed.

6. *Reliability.* This concerns the question how the used algorithms can be verified and how stable the system is. Because an object hierarchy gives a program more structure and because algorithms are usually defined on the abstraction level they belong to, it is easier to verify the code. Error situations as division by zero can be handled by calling methods of an "error object" which can be instructed to behave in a well defined way as to write an error message or to terminate the program.

   Another aspect concerning reliability is that code fragments which are often accessed by many different structures are more reliable then code which is seldom called. In the example of the field hierarchy this means that if e.g. the abstract written powering method has been successfully used for finite fields it is very likely that the powering method for algebraic number fields will also be correct since the code is the same.

7. *Simplicity:* Perhaps the most important demand is that a program has to be simple to use. Concerning mathematical programming this means that the implemented structures have to follow the mathematical intuition. The examples in Section 2 show that inheritance and polymorphism are well suited for this task.

References

[1] T. Budd. *An Introduction to Object-Oriented Programming.* Addison-Wesley, 1998.

[2] CAGE. *Classes for Algebraic Geometry and Elliptic Curves*, http://cage.math.uni-sb.de.

[3] M. Conrad, S. Schmitt. *An object oriented concept for elliptic curves*, submitted to J. Symbolic Comput.

[4] LiDIA. *A C++ library for computational number theory*, http://www.informatik.tu-darmstadt.de/TI/LiDIA. See also [5].

[5] Th. Papanikolaou. *Entwurf und Entwicklung einer objektorientierten Bibliothek für algorithmische Zahlentheorie*. Dissertation an der Universität des Saarlandes, Saarbrücken, 1997.

[6] SIMATH. *A computer algebra system for algorithmic number theory*, http://simath.math.uni-sb.de.

Marc Conrad
Gärtnerstraße 5
D-66117 Saarbrücken
Germany
marc@math.uni-sb.de
http://emmy.math.uni-sb.de/~marc